"Express Mail" mailing label number:

EV 335 379 397 US

OBSTRUCTION-FREE SYNCHRONIZATION FOR SHARED DATA STRUCTURES

Mark S. Moir,
Victor Luchangco and
Maurice Herlihy

CROSS-REFERENCE TO RELATED APPLICATION(S)

[1001] This application claims priority, under 35 U.S.C. § 119(e), of U.S. Provisional Application No. 60/396,152, filed 16 July 2002, naming Mark Moir, Victor Luchangco and Maurice Herlihy as inventors.

BACKGROUND

Field of the Invention

[1002] The present invention relates generally to coordination amongst execution sequences in a multiprocessor computer, and more particularly, to structures and techniques for facilitating nonblocking implementations of shared data structures.

Description of the Related Art

[1003] A traditional way to implement shared data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. However, locking has a number of disadvantages with respect to software engineering, fault-tolerance, and scalability. As a result, researchers have investigated a variety of alternative nonblocking synchronization techniques that do not employ mutual exclusion. A synchronization technique is said to be *wait-free* if it ensures that every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads. It is said to be *lock-free* if it ensures only that some thread always makes progress. While wait-free synchronization is the ideal behavior (thread starvation is unacceptable), lock-free synchronization is often good enough for practical purposes (as long as starvation, while possible in principle, never happens in practice).

[1004] In the hands of a highly skilled programmer, the synchronization primitives provided by many modern processor architectures, such as *compare-and-swap* (CAS) operations or *load-locked/store-conditional* (LL/SC) operation pairs, are typically powerful enough to achieve wait-free (or lock-free) implementations of a linearizable data object. Nevertheless, with a few exceptions, wait-free and lock-free data structures are rarely used in practice. The underlying problem is that conventional synchronization primitives such as CAS and LL/SC are an awkward match for lock-free synchronization. These primitives lend themselves most naturally to *optimistic* synchronization, which guarantees progress only in the absence of synchronization conflicts. For example, the natural way to use CAS for synchronization is to read a value *v* from an address *a*, perform a multistep computation to derive a new value *w*, and then to call CAS to reset the value of *a* from *v* to *w*. The CAS is successful if the value at *a* has not been changed in the meantime. Progress guarantees typically rely on complex and computationally expensive "helping" mechanisms that pose a substantial barrier to the wider use of lock-free synchronization.

[1005] Accordingly, alternative techniques are desired whereby these complexities and related computational expense may be avoided or reduced. In this way, nonblocking shared data objects may achieve wider adoption and use.

SUMMARY

[1006] We propose an alternative nonblocking condition that we believe will, in practice, lead to simple, efficient non-blocking implementations of shared data structures and associated algorithms. Our techniques build on the concept of *obstruction-freedom*. A synchronization technique is *obstruction-free* if it guarantees progress for any thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, a thread is considered to execute in isolation as long as the other threads do not take any steps. Pragmatically, it is enough for the thread to run long enough without encountering a synchronization conflict from a concurrent thread. Like the wait-free and lock-free conditions, obstruction-free synchronization ensures that no thread can be blocked by delays or failures of other threads. This property is weaker than lock-free synchronization,

because it does not guarantee progress when two or more conflicting threads are executing concurrently.

[1007] A somewhat unconventional aspect of our approach of implementing obstruction-free algorithms (which differs from the usual approach of implementing their lock-free and wait-free counterparts) is that we think that progress should be considered a problem of engineering, not of mathematics. We believe that conventional approaches which tend to commingle correctness and progress have inadvertently resulted in unnecessarily inefficient and conceptually complex algorithms, creating a barrier to widespread acceptance of nonblocking forms of synchronization. We believe that a clean separation between the two concerns promises simpler, more efficient, and more effective algorithms.

BRIEF DESCRIPTION OF THE DRAWINGS

[1008] The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

[1009] **FIG. 1** depicts an illustrative state of an array-based encoding of a shared data structure that serves as a context for explaining some techniques in accordance with the present invention.

[1010] **FIG. 2** illustrates a flow for an illustrative *obstruction-free* push operation in accordance with some techniques of the present invention.

[1011] **FIG. 3** depicts an illustrative state of a circular array-based encoding of a shared data structure encoding for an exemplary non-blocking deque implemented in accordance with some embodiments of the present invention.

[1012] **FIG. 4** illustrates a flow for an illustrative *obstruction-free* push operation on a wraparound deque in accordance with some techniques of the present invention.

[1013] The use of the same reference symbols in different drawings indicates similar or identical items.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

[1014] To illustrate the power of our *obstruction-free* approach, we have implemented a nonblocking double-ended queue (i.e., a deque). Deques are more formally defined below. However, informally, deques generalize FIFO queues and LIFO stacks by supporting a sequence of values and operations for adding (pushing) a value to or removing (popping) a value from either end. Thus, implementing a shared deque combines the intricacies of implementing queues and stacks.

[1015] Using our techniques, we believe that we have achieved the first fully-functional, single-target synchronization based (e.g., CAS-based), non-blocking deque implementation in which opposing end operations do not always interfere. To contrast our results with that of others, we briefly summarize related work on nonblocking deques.

[1016] Arora, *et al.* proposed a limited-functionality CAS-based lock-free deque implementation (*See* N. S. Arora, B. Blumofe, and C. G. Plaxton, *Thread Scheduling for Multiprogrammed Multiprocessors*, In Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 119–129 (1998)). Their deque allows only one process to access one end, and only pop operations to be done on the other. Thus, they did not face the difficult problem of concurrent pushes and pops on the same end of the deque. They further simplified the problem by allowing some concurrent operations to simply abort and report failure.

[1017] Greenwald proposed two lock-free deque implementations. *See* M. Greenwald. *Non-Blocking Synchronization and System Design*, PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999. Both implementations depend on a hardware DCAS (double compare-and-swap) instruction, which is not widely supported in practice, and one of them does not support noninterfering concurrent operations at opposite ends of the deque.

[1018] Michael proposed a simple and efficient lock-free, CAS-based deque implementation. *See* M. Michael, *Dynamic Lock-Free Deques Using Single Address, Double-Word CAS*, Technical report, IBM TJ Watson Research Center, January 2002. However, the technique used by Michael's proposed algorithm fundamentally causes

all operations to interfere with each other. Therefore, it offers no insight into designing scalable nonblocking data structures in which noninterfering operations can proceed in parallel.

Obstruction-Free Implementations

[1019] We now introduce obstruction-freedom, a new nonblocking property for shared data structure implementations. This property is strong enough to avoid the problems associated with locks, but it is weaker than previous nonblocking properties—specifically lock-freedom and wait-freedom—allowing greater flexibility in the design of efficient implementations. Obstruction-freedom admits substantially simpler implementations, and we believe that in practice it can provide the benefits of wait-free and lock-free implementations. To illustrate the benefits of obstruction-freedom, we present two obstruction-free CAS-based implementations of double-ended queues (deques); the first is implemented on a linear array, the second on a circular array. To our knowledge, all previous nonblocking deque implementations (i) have been based on unrealistic assumptions about hardware support for synchronization, (ii) have restricted functionality and/or (iii) have operations that interfere with operations at the opposite end of the deque even when the deque has many elements in it. Our obstruction-free implementations exhibit none of these drawbacks. While this is an important achievement in and of itself, the simplicity of our implementations tends to suggest that it is much easier to design obstruction-free implementations than lock-free and wait-free ones.

[1020] Because obstruction-freedom does not guarantee progress in the presence of contention, we need to provide some mechanism to reduce the contention so that progress is achieved. However, lock-free and wait-free implementations typically also require such mechanisms to get satisfactory performance. We can use these same or similar mechanisms with obstruction-free implementations, as we discuss below. Because obstruction-freedom guarantees safety regardless of the contention, we can change mechanisms, even dynamically, without changing the underlying nonblocking implementation.

[1021] One simple and well-known method to reduce contention is for operations to "back off" when they encounter interference by waiting for some time before

retrying. Various choices are possible for how long to wait. For example, randomized exponential backoff is one scheme that is effective in many contexts. Other approaches to reducing contention include queuing and timestamping approaches, in which threads agree amongst themselves to "wait" for each other to finish. While a simplistic applications of these ideas could give rise to some of the same problems that the use of locks does, we have much more freedom in designing sophisticated approaches for contention control than when using locks, because correctness is not jeopardized by interrupting an operation at any time and allowing another operation to continue execution.

[1022] In fact, it is possible to design contention management mechanisms that guarantee progress to every operation that takes enough steps, provided the system satisfies some very weak (and reasonable) assumptions. Thus, the strong progress properties of wait-free implementations can be achieved in practice by combining obstruction-free implementations with appropriate contention managers. In scenarios in which contention between operations is rare, we will benefit from the simple and efficient obstruction-free designs; the more heavy-weight contention resolution mechanisms will rarely be invoked. In contrast, in most lock-free and wait-free implementations, the mechanisms that are used to ensure the respective progress properties impose significant overhead *even in the absence of contention.*

[1023] In some contexts, explicit contention reduction mechanisms may even be unnecessary. For example, in a uniprocessor where threads are scheduled by time slice, relatively short obstruction-free operations will be guaranteed to run alone for long enough to complete. Similarly, in priority-scheduled uniprocessors, an operation runs in isolation unless it is preempted by a higher priority operation.

A Double-ended Queue (Deque)

[1024] A deque object S is a concurrent shared object, that in an exemplary realization is created by an operation of a constructor operation, e.g., `make_deque()`, and which allows each processor P_i, $0 \leq i \leq n-1$, of a concurrent system to perform the following types of operations on S: `push_right`$_i$`(v)`, `push_left`$_i$`(v)`, `pop_right`$_i$`()`, and `pop_left`$_i$`()`. Each push operation has an input, v, where v is selected from a range of values. Each pop operation returns an output from the range

of values. Push operations on a full deque object and pop operations on an empty deque object return appropriate indications.

[1025] A concurrent implementation of a deque object is one that is linearizable to a standard sequential deque. This sequential deque can be specified using a state-machine representation that captures all of its allowable sequential histories. These sequential histories include all sequences of push and pop operations induced by the state machine representation, but do not include the actual states of the machine. In the following description, we abuse notation slightly for the sake of clarity.

[1026] The state of a deque is a sequence of items $S = \langle v_0, \ldots, v_k \rangle$ from the range of values, having cardinality $0 \leq |S| \leq$ `max_length_S`. The deque is initially in the empty state (following invocation of `make_deque()`), that is, has cardinality 0, and is said to have reached a full state if its cardinality is `max_length_S`.

[1027] The four possible push and pop operations, executed sequentially, induce the following state transitions of the sequence $S = \langle v_0, \ldots, v_k \rangle$, with appropriate returned values:

`push_right(`v_{new}`)`	if S is not full, sets S to be the sequence $S = \langle v_0, \ldots, v_k, v_{new} \rangle$
`push_left(`v_{new}`)`	if S is not full, sets S to be the sequence $S = \langle v_{new}, v_0, \ldots, v_k \rangle$
`pop_right()`	if S is not empty, sets S to be the sequence $S = \langle v_0, \ldots, v_{k-1} \rangle$ and returns the item, v_k.
`pop_left()`	if S is not empty, sets S to be the sequence $S = \langle v_1, \ldots, v_k \rangle$ and returns the item v_0.

[1028] For example, starting with an empty deque state, $S = \langle\rangle$, the following sequence of operations and corresponding transitions can occur. A `push_right(1)` changes the deque state to $S = \langle 1 \rangle$. A `push_left(2)` subsequently changes the deque state to $S = \langle 2,1 \rangle$. A subsequent `push_right(3)` changes the deque state to $S = \langle 2,1,3 \rangle$. Finally, a subsequent `pop_right()` changes the deque state to $S = \langle 2,1 \rangle$ and returns the value, 3. In some implementations, return values may be employed to indicate success or failure.

Obstruction-Free Deque Implementation

[1029] We present an array-based, obstruction-free deque implementation. Our first algorithm is extremely simple, and serves to illustrate our technique. However, the first illustration is not entirely complete in the sense that it does not fully generalize queues. In particular, if we only push on one end and pop from the other, we will exhaust the space in the illustrated array and will not be able to push any more items. Later, we show how to extend the algorithm to "wrap around" in the array in order to overcome this problem.

[1030] The declarations that follow define a simple array-based data structure that encodes our deque.

```
type element = record val: valtype; ctr: int end
A: array[0..MAX+1] of element initially there is some k in
[0,MAX] such that A[i]=<LN,0> for all i in [0,k] and
A[i]=<RN,0> for all i in [k+1,MAX+1].
```

[1031] In our implementation, we assume the existence of two special "null" values `LN` and `RN` (left null and right null) that are never pushed onto the deque. We use the array `A` to store the current state of the deque. The deque can contain up to `MAX` values, and the array is of size `MAX+2` to accommodate a left-most location that always contains `LN` and a right-most location that always contains `RN`. These extra locations are not strictly necessary, but they simplify the code.

[1032] **FIG. 1** illustrates such an array **110**, where values v_1, v_2, ... v_n of a represented deque are encoded in elements of the array. An `LN` value is stored in a leftmost array element **111** and in each other element to the *left* of v_1. An `RN` value is stored in a rightmost array element **112** and in each other element to the *right* of v_n. Each element of the array includes two fields, e.g., a `val` field such as **113A** and a `ctr` field such as **113B**. Operations on the encoded values and on fields of the elements will be understood with reference to **FIG. 1** and to the exemplary code herein.

[1033] Our algorithm maintains the invariant that the sequence of values in `A[0].val..A[MAX+1].val` always includes of at least one `LN`, followed by zero or more data values, followed by at least one `RN`. The array can be initialized any way

that satisfies this invariant. To simplify our presentation, we assume the existence of a function `oracle()`, which accepts a parameter `left` or `right` and returns an array index. The intuition is that this function attempts to return the index of the leftmost `RN` value in `A` when invoked with the parameter `right`, and attempts to return the index of the rightmost `LN` value in `A` when invoked with the parameter `left`. The algorithm is linearizable even if `oracle` can be incorrect. We assume that `oracle()` always returns a value between 1 and `MAX+1`, inclusive, when invoked with the parameter `right` and always returns a value between 0 and `MAX`, inclusive, when invoked with the parameter `left`. Clearly, it is trivial to implement a function that satisfies this property. Stronger properties of the oracle are required to prove obstruction-freedom; we discuss these properties and how they can be achieved later.

[1034] As explained in more detail below, we employ version numbers to each value in order to prevent concurrent operations that potentially interfere from doing so. The version numbers are updated atomically with the values using a compare-and-swap (CAS) instruction. In general, a `CAS(a,e,n)` operation or instruction takes three parameters: an address `a`, an expected value `e`, and a new value `n`. If the value currently stored at address `a` matches the expected value `e`, then the `CAS` stores the new value `n` at address `a` and returns *true;* we say that the `CAS` *succeeds* in this case. Otherwise, the `CAS` returns *false* and does not modify the memory. We say that the `CAS` *fails* in this case. As usual with version numbers, we assume that sufficient bits are allocated for the version numbers to ensure that they cannot "wrap around" during the short interval in which one process executes a single iteration of a short loop in our algorithm.

[1035] A reason our obstruction-free deque implementation is so simple, and the reason we believe obstruction-free implementations in general will be significantly simpler than their lock-free and wait-free counterparts, is that there is no progress requirement when interference is detected. Thus, provided we maintain basic invariants, we can simply retry when we detect interference. In our deque implementation, data values are changed only at the linearization point of successful push and pop operations. To detect when concurrent operations interfere with each other, we increment version numbers of adjacent locations (without changing their

associated data values). As a result of this technique, two concurrent operations can each cause the other to retry: this explains why our implementation is so simple, and also why it is not lock-free.

[1036] To make this idea more concrete, we describe our implementation in terms of right-side push and pop operations (`rightpush()` and `rightpop()`) that appear below. Left-side operations are symmetric with the right-side ones. As a result, they are not separately described.

```
rightpush(v)                  // v is not RN or LN
RH0: while (true) {
RH1:   k := oracle(right);    // find index of leftmost RN
RH2:   prev := A[k-1];        // read (supposed) rightmost non-RN
                              // value
RH3:   cur := A[k];           // read (supposed) leftmost RN value
RH4:   if (prev.val != RN and cur.val = RN) { // oracle is right
RH5:     if (k = MAX+1) return "full";        // A[MAX] != RN
RH6:     if CAS(&A[k-1],prev,<prev.val,prev.ctr+1>) // try to
                                                    // bump up
                                                    // prev.ctr
RH7:       if CAS(&A[k],cur,<v,cur.ctr+1>)    // try to push new
                                              // value
RH8:         return "ok";  // it worked!
       }
     }
```

[1037] **FIG. 2** highlights the simplicity of our implementation. After consulting **(201)** an oracle (described below), the `rightpush()` implementation employs a pair of single-target synchronizations to increment **(202)** the `ctr` field of an adjacent element and, if successful, to push **(203)** a value onto the deque. Interference with a competing operation simply results in a retry.

[1038] The `rightpop()` operation is also quite straightforward and will be understood with reference to the following code.

```
rightpop()
RP0: while (true) {    // keep trying till return val or empty
RP1:   k := oracle(right);  // find index of leftmost RN
RP2:   cur := A[k-1];       // read (supposed) value to be popped
RP3:   next := A[k];        // read (supposed) leftmost RN
RP4:   if (cur.val != RN and next.val = RN) { // oracle is right
RP5:     if (cur.val = LN and A[k-1] = cur); // adjacent LN and RN
RP6:       return "empty"
RP7:     if CAS(&A[k],next,<RN,next.ctr+1>) // try to bump up
                                            // next.ctr
RP8:       if CAS(&A[k-1],cur,<RN,cur.ctr+1>) // try to remove
value
RP9:         return cur.val  // it worked; return removed value
       }
     }
```

[1039] The basic idea behind our algorithm is that a `rightpush(v)` operation changes the leftmost `RN` value to `v`, and a `rightpop()` operation changes the rightmost data value to `RN` and returns that value. Each `rightpush(v)` operation that successfully pushes a data value (as opposed to returning "full") is linearized to the point at which it changes an `RN` value to `v`. Similarly, each `rightpop()` operation that returns a value `v` (as opposed to returning "empty") is linearized to the point at which it changes the `val` field of some array location from `v` to `RN`. Furthermore, the `val` field of an array location does not change unless an operation is linearized as discussed above. The `rightpush()` operation returns "full" only if it observes a non-`RN` value in `A[MAX].val`. Given these observations, it is easy to see that our algorithm is linearizable if we believe the following three claims (and their symmetric counterparts):

- At the moment that line `RH7` of a `rightpush(v)` operation successfully changes `A[k].val` for some `k` from `RN` to `v`, `A[k-1].val` contains a non-`RN` value (i.e., either a data value or `LN`).
- At the moment that line `RP8` of the `rightpop()` operation successfully changes `A[k-1].val` for some `k` from some value `v` to `RN`, `A[k].val` contains `RN`.
- If a `rightpop()` operation returns "empty", then at the moment it executed line `RP3`, `A[k].val=RN` and `A[k-1].val=LN` held for some `k`.

[1040] Using the above observations and claims, a proof by simulation to an abstract deque in an array of size `MAX` is straightforward. Below we briefly explain the synchronization techniques that we use to ensure that the above claims hold. The techniques all exploit the version numbers in the array locations.

[1041] The empty case (the third claim above) is the simplest: `rightpop()` returns "empty" only if it reads the same value from `A[k-1]` at lines `RP2` and `RP5`. Because every CAS that modifies an array location increments that location's version number, it follows that `A[k-1]` maintained the same value throughout this interval (recall our assumption about version numbers not wrapping around). Thus, in particular, `A[k-1].val` contained `LN` at the moment that line `RP3` read `RN` in `A[k].val`.

[1042] The techniques used to guarantee the other two claims are essentially the same, so we explain only the first one. The basic idea is to check that the neighboring location (i.e., `A[k-1]`) contains the appropriate value (line `RH2`; see also line `RH4`), and to increment its version number (without changing its value; line `RH6`) between reading the location to be changed (line `RH3`) and attempting to change it (line `RH7`). If any of the attempts to change a location fail, then we have encountered some interference, so we can simply restart. Otherwise, it can be shown easily that the neighboring location did not change to `RN` between the time it was read (line `RH2`) and the time the location to be changed is changed (line `RH7`). The reason is that a `rightpop()` operation—the only operation that changes locations to `RN`—that was attempting to change the neighboring location to `RN` would increment the version number of the location the `rightpush()` operation is trying to modify, so one of the operations would cause the other to retry.

Oracle Implementations

[1043] The requirements for the `oracle()` function assumed in the previous section are quite weak, and therefore a number of implementations are possible. We first describe the requirements, and then outline some possible implementations. For linearizability, the only requirement on the oracle is that it always returns an index from the appropriate range depending on its parameter as stated earlier; satisfying this requirement is trivial. However, to guarantee obstruction-freedom, we require that the oracle is *eventually accurate if repeatedly invoked in the absence of interference*. By "accurate": we mean that it returns the index of the leftmost `RN` when invoked with right, and the index of the rightmost `LN` when invoked with `left`. It is easy to see that if any of the operations executes an entire loop iteration in isolation, and the oracle

function returns the index specified above, then the operation completes in that iteration. Because the oracle has no obligation (except for the trivial range constraint) in the case that it encounters interference, we have plenty of flexibility in implementing it. One simple and correct implementation is to search the array linearly from one end looking for the appropriate value. Depending on the maximum deque size, however, this solution might be very inefficient. One can imagine several alternatives to avoid this exhaustive search. For example, we can maintain "hints" for the left and right ends, with the goal of keeping the hints approximately accurate; then we could read those hints, and search from the indicated array position (we'll always be able to tell which direction to search using the values we read). Because these hints do not have to be perfectly accurate at all times, we can choose various ways to update them. For example, if we use CAS to update the hints, we can prevent slow processes from writing out-of-date values to hints, and therefore keep hints almost accurate all the time. It may also be useful to loosen the accuracy of the hints, thereby synchronizing on them less often. In particular, we might consider only updating the hint when it is pointing to a location that resides in a different cache line than the location that really contains the leftmost `RN` for example, as in this case the cost of the inaccurate hint would be much higher.

Extension to Circular Arrays

[1044] In this section, we show how to extend the algorithm in the previous section to allow the deque to "wrap around" the array, so that the array appears to be circular. **FIG. 3** illustrates a two-value deque state encoded in a suitable circular array (**300**), where elements **301** and **302** encode values v_1 and v_2, respectively. In other words, `A[0]` is "immediately to the right" of `A[MAX+1]`. As before, we maintain at least two null entries in the array: we use the array `A[0..MAX+1]` for a deque with at most `MAX` elements. The array can be initialized arbitrarily provided it satisfies the main invariant for the algorithm, stated below. One option is to use the initial conditions for the algorithm in the previous section.

[1045] We now describe the new aspects of the algorithm. Code for the right-side operations of the wrap-around deque implementation are shown below. As before, the left-side operations are symmetric, and we do not discuss them further except as

they interact with the right-side operations. All arithmetic on array indices is done modulo MAX+2.

```
rightpush(v)                                          // !(v in {LN,RN,DN})
RH0: while (true) {
RH1:   k,prev,cur := rightcheckedoracle();  // cur.val = RN and
                                                      // prev.val != RN
RH2:   next := A[k+1];
RH3:   if (next.val = RN)
RH4:     if CAS(&A[k-1], prev, <prev.val, prev.ctr+1>)
RH5:       if CAS(&A[k], cur, <v, cur.ctr+1>)        // RN -> v
RH6:         return "ok";
RH7:   if (next.val = LN)
RH8:     if CAS(&A[k], cur, <RN, cur.ctr+1>)
RH9:       CAS(&A[k+1], next, <DN, next.ctr+1>);     // LN -> DN
RH10:  if (next.val = DN) {
RH11:    nextnext := A[k+2];
RH12:    if !(nextnext.val in {RN,LN,DN})
RH13:      if (A[k-1] = prev)
RH14:        if (A[k] = cur) return "full";
RH15:    if (nextnext.val = LN)
RH16:      if CAS(&A[k+2], nextnext, <nextnext.val,nextnext.ctr+1>)
RH17:        CAS(&A[k+1], next, <RN, next.ctr+1>);  // DN -> RN
       }
     }

rightpop()
RP0: while (true) {
RP1:   k,cur,next := rightcheckedoracle();  // next.val = RN and
                                                      // cur.val != RN
RP2:   if (cur.val in {LN,DN} and A[k-1] = cur) // depends on order
                                                      // of RO2 & RO3.
RP3:     return "empty";
RP4:   if CAS(&A[k],next,<RN,next.ctr+1>)
RP5:     if CAS(&A[k-1],cur,<RN,cur.ctr+1>)      // v -> RN
RP6:       return cur.val;
     }
```

[1046] There are two main differences between this algorithm and the one in the previous section. First, it is more difficult to tell whether the deque is full; we must determine that there are exactly two null entries. Second, `rightpush()` operations may encounter LN values as they "consume" the RN values and wrap around the array (similarly, `leftpush()` operations may encounter RN values). We handle this second problem by enabling a `rightpush()` operation to "convert" LN values into RN values. This conversion uses an extra null value, which we denote DN, for "dummy null". We assume that LN, RN and DN are never pushed onto the deque.

[1047] Because the array is circular, the algorithm maintains the following invariants instead of the simpler invariant maintained by the algorithm in the previous section:

- All null values are in a contiguous sequence of locations in the array. (Recall that the array is circular, so the sequence can wrap around the array.)
- The sequence of null values consists of zero or more RN values, followed by zero or one DN value, followed by zero or more LN values.
- There are at least two different types of null values in the sequence of null values.

Thus, there is always at least one LN or DN entry, and at least one RN or DN entry.

[1048] Instead of invoking `oracle(right)` directly, the push and pop operations invoke a new auxiliary procedure, `rightcheckedoracle()`. In addition to an array index `k`, `rightcheckedoracle()` returns `left` and `right`, the contents it last saw in `A[k-1]` and `A[k]` respectively. It guarantees that `right.val=RN` and that `left.val!=RN`. Thus, if `rightcheckedoracle()` runs in isolation, it always returns the correct index, together with contents of the appropriate array entries that prove that the index is correct. If no RN entry exists, then by the third invariant above, there is a DN entry and an LN entry; `rightcheckedoracle()` attempts to convert the DN into an RN before returning.

```
rightcheckedoracle()
// Returns k,left,right, where left = A[k-1] at some time t,
// and right = A[k] at some time t' > t during the execution,
// with left.val != RN and right.val = RN.
RO0: while (true) {
RO1:   k := oracle(right);
RO2:   left := A[k-1];                          // order important for check
RO3:   right := A[k];                           // for empty in rightpop
RO4:   if (right.val = RN and left.val != RN) // correct oracle
RO5:     return k,left,right;
RO6:   if (right.val = DN and !(left.val in {RN,DN})) // correct oracle,
                                                      // but no RNs
RO7:     if CAS(&A[k-1], left, <left.val,left.ctr+1>)
RO8:       if CAS(&A[k], right, <RN,right.ctr+1>)     // DN -> RN
RO9:         return k,<left.val,left.ctr+1>,<RN,right.ctr+1>;
     }
```

[1049] Other than calling `rightcheckedoracle()` instead of `oracle(right)`, which also eliminates the need to read and check the `cur` and `next` values again, the only change in the `rightpop()` operation is that, in checking whether the deque is empty, `cur.val` may be either LN or DN, because there may be no LN entries.

[1050] **FIG. 4** summarizes major flows in the operation of the `rightpush()` and `rightcheckedoracle()` operations. As before, the implementation consults **(401)**

an oracle, though this time, we check for the possibility that (though correct) the oracle returns a location with an adjacent DN value encoding to be converted. If so, the `rightpush()` operation employs a sequence of synchronization operations (at **402**) to perform the DN to RN conversion. If successful (or if no DN conversion was required), the `rightpush()` operation attempts (as before) to increment an appropriate `ctr` field and to update a corresponding value field using a simple sequence of single-target (e.g., CAS-based) synchronization operations. As before, on failure, we simply retry. Though structurally quite similar to the simple example described above, our circular array algorithm does differentiate (at **403**) between various conditions (*space available*, *wrap*, and DN-to-RN *conversion needed*) to perform the appropriate value update (e.g., at **404A**, **404B** or **404C**) as part of the synchronization updates.

[1051] Because the array is circular, a `rightpush()` operation cannot determine whether the array is full by checking whether the returned index is at the end of the array. Instead, it ensures that there is space in the array by checking that `A[k+1].val=RN`. In that case, by the third invariant above, there are at least two null entries other than `A[k]` (which also contains RN), so the deque is not full. Otherwise, `rightpush()` first attempts to convert `A[k]` into an RN entry. We discuss how this conversion is accomplished below.

[1052] When a `rightpush()` operation finds only one RN entry, it tries to convert the next null entry—we know there is one by the third invariant above—into an RN. If the next null entry is an LN entry, then `rightpush()` first attempts to convert it into a DN entry. When doing this, `rightpush()` checks that `cur.val=RN`, which ensures there is at most one DN entry, as required by the second invariant above. If the next null entry is a DN entry, `rightpush()` will try to convert it into an RN entry, but only if the entry to the right of the one being converted (the `nextnext` entry) is an LN entry. In this case, it first increments the version number of the `nextnext` entry, ensuring the failure of any concurrent `leftpush()` operation trying to push a value into that entry. If the `nextnext` entry is a deque value, then the `rightpush()` operation checks whether the right end of the deque is still at `k` (by rereading `A[k-1]` and `A[k]`), and if so, the deque is full. If not, or if the `nextnext`

entry is either an RN or DN entry, then some other operation is concurrent with the `rightpush()`, and the `rightpush()` operation retries.

[1053] Assuming the invariants above, it is easy to see that this new algorithm is linearizable in exactly the same way as the algorithm in the previous section, except that a `rightpush()` operation that returns "full" linearizes at the point that `nextnext` is read (line RH11). Because we subsequently confirm (line RH13) that `A[k-1]` and `A[k]` have not changed since they were last read, we know the deque extends from `A[k+2]` to `A[k-1]` (with `A[k-1]` as its rightmost value), so that `A[k]` and `A[k+1]` are the only nonnull entries, and thus, the deque is full.

[1054] The main difficulty is verifying that when a `rightpush()` actually pushes the new value onto the deque (line RH5), either the `next` entry is an RN entry, or it is a DN entry and the `nextnext` entry is an LN entry. This is to ensure that after the push, there are still at least two null entries, one of which is an RN or DN entry. One key to the proof is to note that the value of an entry is changed only by lines R08, RH5, RH9, RH17, RP5, and their counterparts in the left-side operations. Furthermore, these lines only change an entry if the entry has not changed since it was most recently read. These lines are annotated with a description of how they change the value of the entry.

[1055] *Time complexity.* A simple measure of the time complexity of an obstruction-free algorithm (without regard to the particular contention manager and system assumptions) is the worst-case number of steps that an operation must take in isolation in order to be guaranteed to complete. For our algorithms, this is a constant plus the obstruction-free time complexity of the particular oracle implementation used.

Other Embodiments

[1056] The description presented herein includes a set of techniques, objects, functional sequences and data structures associated with concurrent shared object implementations employing linearizable synchronization operations in accordance with an exemplary embodiment of the present invention. An exemplary non-blocking, linearizable concurrent double-ended queue (deque) implementation that

employs compare-and-swap (CAS) operations is illustrative. As described, our implementation is obstruction-free. The deque is a good exemplary concurrent shared object implementation in that it involves all the intricacies of LIFO-stacks and FIFO-queues, with the added complexity of handling operations originating at both of the deque's ends. Accordingly, techniques, objects, functional sequences and data structures presented in the context of a concurrent deque implementation will be understood by persons of ordinary skill in the art to describe a superset of support and functionality suitable for less challenging concurrent shared object implementations, such as LIFO-stacks, FIFO-queues or concurrent shared objects (including deques) with simplified access semantics.

[1057] While the invention(s) is(are) described with reference to various implementations and exploitations, it will be understood that these embodiments are illustrative and that the scope of the invention(s) is not limited to them. Terms such as always, never, all, none, etc. are used herein to describe sets of consistent states presented by a given computational system, particularly in the context of correctness proofs. Of course, persons of ordinary skill in the art will recognize that certain transitory states may and do exist in physical implementations even if not presented by the computational system. Accordingly, such terms and invariants will be understood in the context of consistent states presented by a given computational system rather than as a requirement for precisely simultaneous effect of multiple state changes. This "hiding" of internal states is commonly referred to by calling the composite operation "atomic", and by allusion to a prohibition against any process seeing any of the internal states partially performed.

[1058] Many variations, modifications, additions, and improvements are possible. For example, while application to particular concurrent shared objects and particular implementations thereof have been described in detail herein, applications to other shared objects and other implementations will also be appreciated by persons of ordinary skill in the art. In addition, more complex shared object structures may be defined, which exploit the techniques described herein. While much of description herein has focused on compare and swap (CAS) based synchronization, other synchronization primitives may be employed. For example, based on the description herein, persons of ordinary skill in the art will appreciate that other suitable

constructs, including load-linked and store-conditional operation pairs (LL/SC) may be employed, as well. Plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of the invention(s).

[1059] In general, structures and functionality presented as separate components in the exemplary configurations may be implemented as a combined structure or component. Similarly, structures and functionality presented as a single component may be implemented as separate components. These and other variations, modifications, additions, and improvements may fall within the scope of the invention(s).